1-1-2004

Query, indexing and benchmark for XML-based bibliography databases

Chunrong Pan
Iowa State University

Query, indexing and benchmark for XML-based bibliography databases

by

Chunrong Pan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Shashi Gadia, Major Professor
Douglas Jacobson
Simanta Mitra

Iowa State University

Ames, Iowa

2004

Graduate College
Iowa State University

This is to certify that the master's thesis of

Chunrong Pan

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

Table of Contents

1. Introduction

XML is becoming the de facto standard for data exchange over the Internet. In particular, many XML database contains a large set of small XML documents with similar structure. It creates a new requirement to store and retrieve information from these XML documents efficiently. XML-based bibliography file is such a large set of XML documents with similar structure. The problem we are concerned is to build index and support query for this kind of XML documents in a fast and effective way. Since each document is small, the query result is not necessarily a tree or subtree of XML document. We can return the document ID for each query in order to retrieve the entire document. Document ID is a pointer-like structure and therefore the document can be retrieved from backend storage.

In order to solve this problem, we propose to store a set of XML documents into B+ tree inverted files and query the information based on B+ tree structure. This project uses XML as a bridge and combines the database and information retrieval into one application with supporting storage, indexing and querying. It allows creating index based on the entire path or keywords so that it could retrieval the document ID for keyword query and path query. Based on the assumption that the document is small, this project can deal with the queries with simple path structure or terms very efficiently.

1.1 XML

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is a simple, very flexible text format derived from SGML (ISO 8879) [1]. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [2].

XML provides a standard syntax for the markup of data and documents and also provides a mechanism to impose constraints on the storage layout and logical structure. A valid XML document is made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data and some of which form markup. Markup encodes a description of the document's storage layout and logical structure.

XML contains the components called elements, attributes, comments, processing instructions, entity reference and CDATA sections [3]. An element of XML documents consists of a start tag, an end tag, and the information between the tags, which is often referred to as the contents. An attribute is a parameter to an element.

XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. Logically the document is composed of declarations, elements, comments, character references and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly [4].

XML provides a flexible and efficient way for data exchange, data access, data view and data search. In our project, we focus on the XML data usage in the fields of databases and information retrieval.

1.2 Querying in XML

There is an increasing amount of information stored, exchanged and viewed using XML. The ability to navigate and query XML data from diverse sources becomes increasingly critical. There are two languages introduced to provide the support of navigation and query of XML data.

1.2.1 Xpath

XPath is a language for addressing parts of an XML document, designed to be used by both Extensible Stylesheet Language Transformations (XSLT) and XML Pointer Language (XPointer) [5].

It allows the navigation of XML documents, with the purpose of selecting individual attributes, elements or other parts of an XML document for processing. To support this primary purpose, it facilitates the manipulations of strings, numbers and Booleans.

Other than addressing, XPath can test if a node matches the pattern. This matching ability is used with the other two XML-based technologies that are XSLT and XPointer.

XPath models an XML document as a tree of nodes including element nodes, attribute nodes and text nodes. The primary syntactic construct in XPath is the expression and an expression is evaluated to yield an object. The object can be a collection of nodes, boolean, string or number.

1.2.2 XQuery

A query language that uses the structure of XML intelligently can express queries across all kinds of data, whether physically stored in XML or viewed as XML via middleware [6]. This query language is called XQuery. XQuery is the W3C standard query language for XML data, which is designed to be broadly applicable across many types of XML data sources.

XQuery is to provide a powerful and standardized way of searching through all that XML-encapsulated data. It provides a flexible and easy-to-use mechanism for querying not only content but structure as well [7].

The functionality of XPath is greatly needed as part of XQuery. Therefore compatibility with XPath Version 1 was a major objective for the design of XQuery.

1.3 Databases versus Information Retrieval

The field of information retrieval has focused on searching collections of text documents while the database field desires to expand and manage the data in a DBMS. XML resides in the middle between them and has brought these two fields closer together than ever before.

Although database and information retrieval have the common objective of supporting searches over collections of data, one of the major differences between database and information retrieval is that information retrieval systems are designed to support a specialized class of queries that we called "searches" while database systems support a very general class of queries.

Searches are specified in terms of a few search terms and the underlying data in information retrieval systems is usually a collection of unstructured text documents. There are two types of searches called boolean query or ranked query. In a boolean query, the user specifies an expression constructed using terms and boolean operators. However, in a ranked query the user specifies one or more terms, and the result of the query is a list of documents ranked by the relevance to the query. One of important features of Information retrieval is ranked results. That means all the results are ordered based on how well it matches the terms given. In contrast, the underlying documents are rigidly structured in database systems. Query results from database systems don't have any rank.

XML provides a bridge between databases and information retrieval since each XML document can be marked up to indicate additional information for structure or other details, which has changed the

nature of documents from free text to textual objects with associated fields containing metadata (data about data) or descriptive information [8].

1.4 B+ Tree Index

We use B+ tree to construct the inverted index. It allows the fast retrieval of documents from the disk for the given terms.

1.4.1 Inverted Index

An inverted index is a data structure that enables fast retrieval of all documents that contain a query term. For each term, the index maintains a list (called the inverted list) of entries describing occurrences of the term, with one entry per document that contains the term [8].

For example, we have 4 terms, which are "language", "database", "agent" and "network", from document 1, 2, 3 ... and 10. For each term, there is one corresponding pointer pointing to the position on the disk. We call this pointer as document ID. Document ID is the unique number that represents a single document. Therefore if user searches for one term from these four terms, the retrieval can be done very fast since there is a pointer to the physical position of the corresponding document. Figure 1 shows the structure of an inverted index for this example.

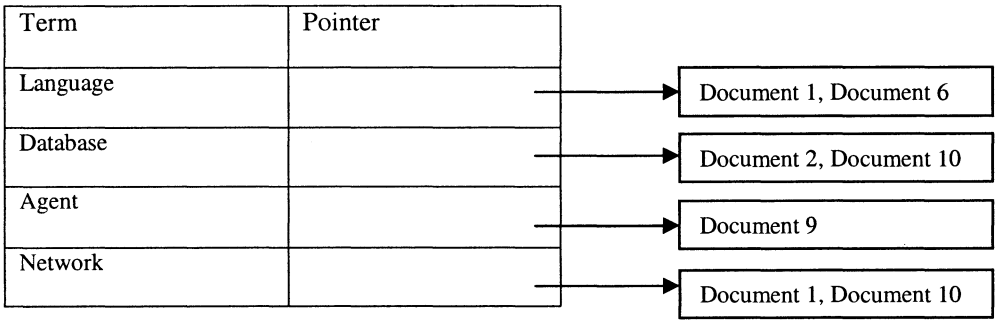

Figure 1: Sample Inverted Index

1.4.2 B+ Tree

A B+ tree has a tree like structure with multi-way fanout from the root down to the leaves, so that the following properties hold [9]:

a) Every node is disk-page sized and resides in a well-defined location on the disk.

b) Nodes above the lead level contain directory entries, with n-1 separate keys and n disk pointers to lower-level B+ tree nodes.

c) Nodes at the leaf level contain entries with real record.

d) All nodes below the root are at least half full with entry information.

A B+ tree consists of an index set and a sequence set. A node in an index set is called an index node and a node in the sequence set is called a sequence nod [10]. As Figure 2 shows, the keys are stored in the index set and records are stored in the sequence set that is the last level of the tree.

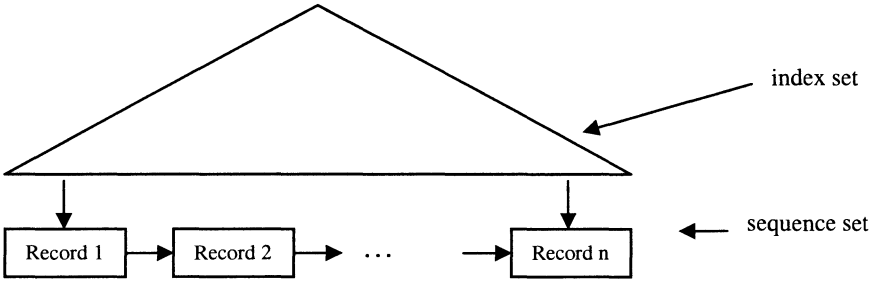

Figure 2: B+ Tree Structure

1.5 Berkeley DB

Berkeley DB is not a database system but a storage technology that can used to build embedded database applications. Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications. It provides a simple function-call application program interface (API) for data access and management [11]. We used Berkeley DB Java API for the purpose of implementations.

Berkeley DB supports four access methods that are hash tables, btrees, simple record-number-based storage and persistent queues. It allows the developers to choose the storage structure that best suits the application.

Hash table uses a hash function that maps key value of a record to an address of blocks on the disk where the record is stored. Berkeley DB uses the technique called extended linear hashing [12]. It gives a good average performance for retrieval of records by the key value in the large database. However hash table method doesn't support range search. That is one of most important reasons we don't use this access method.

Btrees are better for range search. When the application retrieves the data with all the keys started with the given position, btree is the best choice. The tree structure keeps the keys sorted and close to each other in the storage so that sometime fetching the nearby key doesn't require a disk access. This feature is very important to our project. It reduces the number of disk accesses in range search.

Record-number-based storage is well suited for the application that requires storing and retrieving the record without a good way to generate the key for each record. Then the unique number generated automatically by the system can be used as a key.

Queue is best used in the application that operates the data based on the order that are created.

From the comparisons it is obvious that queue and record-number-based storage are not suitable for our project. Therefore, we should not take them into consideration. Between btree access method and hash table access method, we can btree is a better fit for our project.

Berkeley DB is very efficient in compressing the keys which improves the query performance quite a lot. It uses the suffix to distinguish the keys so that it only stores the part of keys instead of the entire keys. The following example shows how this compression is done.

Given the keys "ABC", "CBD" and "CBEF", Berkeley DB stores "A" for the key "ABC", "C" for "CBD" and "CBE" for "CBEF" since "C" is enough to distinguish "ABC" and "CBD" and "CBE" are needed to distinguish "CBD" and "CBEF".

For the rest of this manuscript, we introduce the related concepts in Section 1. We present the related work in Section 2. We describe our design and implementation in Section 3. We implemented three B+ trees: DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree. We use them to do queries and information retrieval searches from large set of bibliography XML documents. We provide the experimental results in Section 4. We discuss the future work in Section 5 and conclude this manuscript in Section 6.

2. Related Works

2.1 RIST [13]

The RIST stands for Relationships Indexed Suffix Tree. RIST is an index method for querying XML data by tree structures. It consists of two types of B+ trees. The first type is the combination of D-Ancestor B+ tree and S-Ancestor B+ trees. The second type is called DocId B+ tree. D-Ancestorship is the ancestor-descendant relationships of the nodes that they represent in the original XML document tree. S-Ancestorship is the ancestor-descendant relationships of the nodes that they represent in the suffix tree. D-Ancestor B+ tree, S-Ancestor B+ tree and suffix tree are explained in the following example.

Figure 3 shows Document 1, Document 2 and the encoded string representations of these two documents.

Doc1:

```
...
  <P>
    <S>
      <N>v1</N>
      <L>v2</L>
    <S>
  </P>
...
```

Encoded String Representation of Doc1:

... (P, ε)(S,P)(N,PS)(v1, PSN)(L,PS)(v2, PSL)

Doc2:

```
...
  <P>
    <B>
      <L>v2</L>
    <B>
  </P>
...
```

Encoded String Representation of Doc2:

... (P, ε)(B,P)(L,PB)(v2, PBL)

Figure 3: Structures and Encoded String Representations of Doc 1 and Doc 2

Figure 4 shows the suffix tree structure for Doc 1 and Doc 2. Doc 1 and Doc 2 share the same root. The left path is tree representation of Doc 1 and the right path is tree representation of Doc 2. For

each Node, it maintains two fields: the sequence of context node and the pair of unique ID with size of descendants.

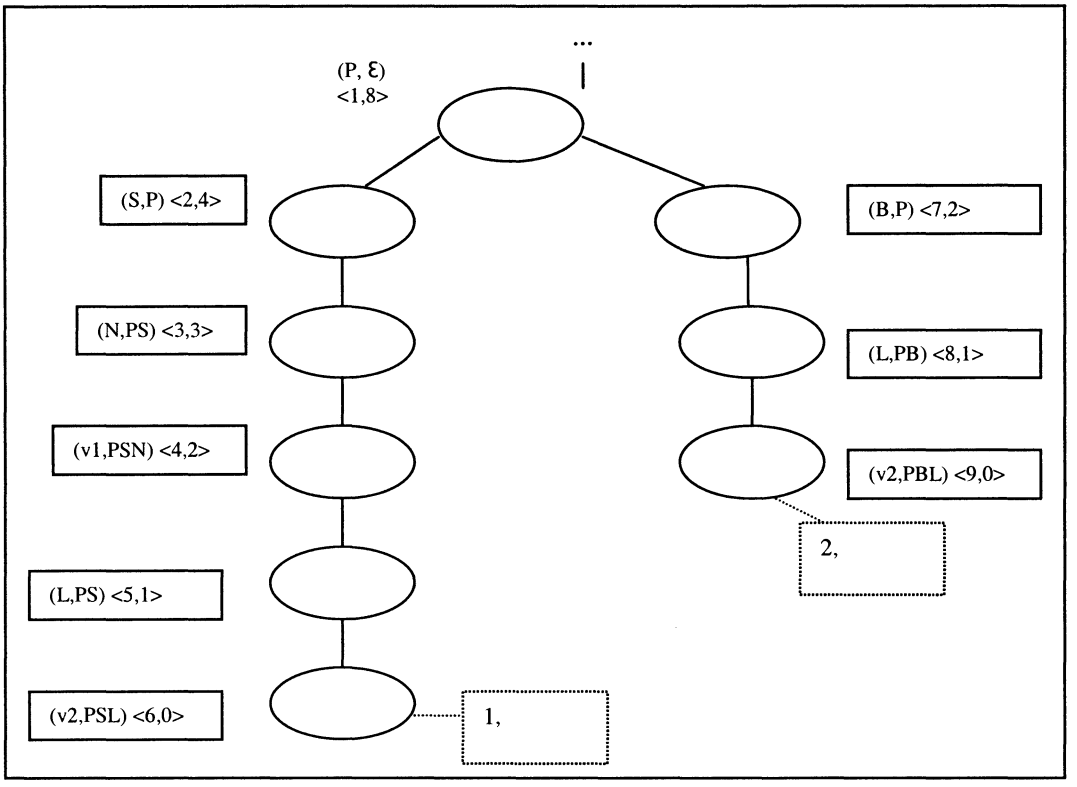

Figure 4: Suffix Tree of Doc 1 and Doc 2

From Figure 4 we can see the element (S, P) is a D-ancestor of (L, PS) and element (v1, PSN) is an S-Ancestor of (L,PS).

With RIST, the next qualified node X after matching a prefix of the query is one of those nodes Y to which X is both a D-Ancestor and an S-ancestor. The D-Ancestorship between two elements can be determined by checking their prefixes. For each node, it has its own labeling $<N_x, Size_x>$. N_x is the prefix traversal order of x in the suffix tree and $Size_x$ is the total number of descendants of x in the suffix tree. With this labeling, S-Ancestorship can be decided easily: if x and y are labeled $<N_x, Size_x>$ and $<N_y, Size_y>$ respectively, node x is an S-Ancestor of y iff $N_y \in (N_x, N_x+Size_x)$.

For D-Ancestor tree, key is the symbol representing the context node plus all the prefixes and value points to the root of an S-Ancestor B+ tree. For each S-Ancestor B+ tree, the key is the ID generated from the preorder tree traversal and the value is ID with size of its descendants.

Figure 5.1 and 5.2 show the detailed structure of RIST. It has two levels of B+ trees. The first level is D-Ancestor tree and the second level is a set of S-Ancestor tree.

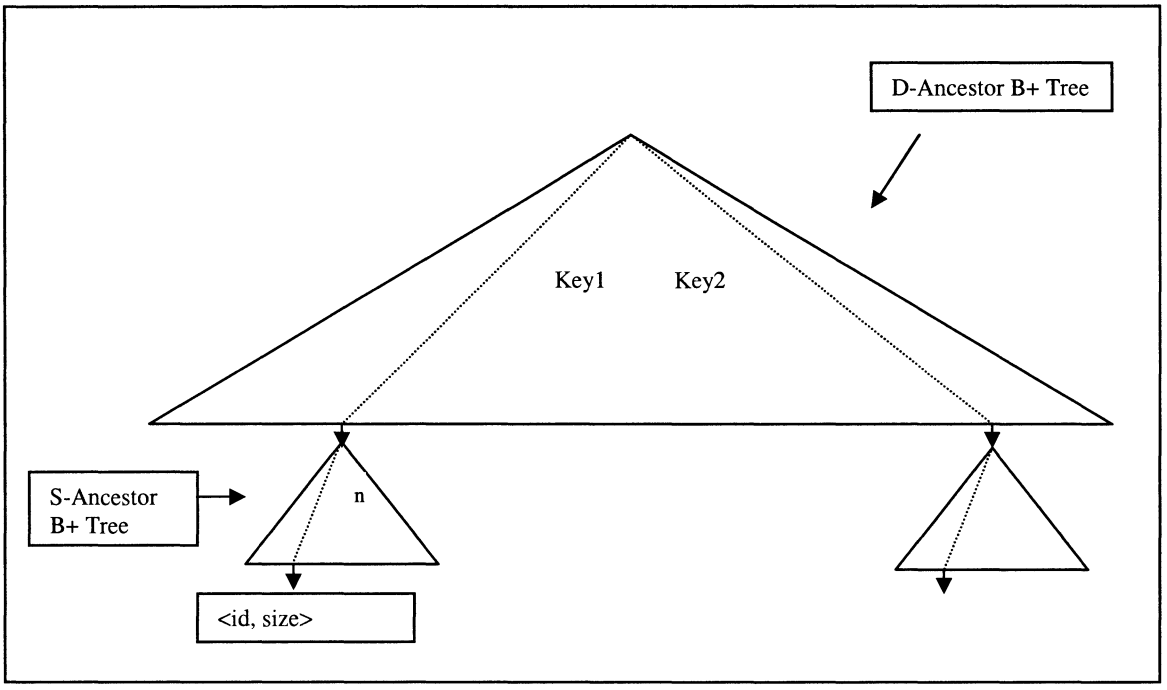

Figure 5.1: Structure of D-Ancestor B+ Tree and S-Ancestor B+ Trees

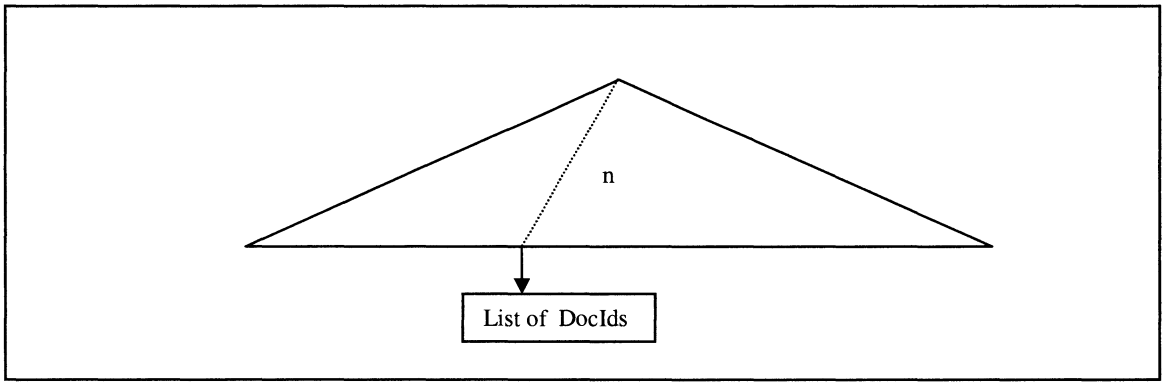

Figure 5.2: Structure of DocId B+ Tree

The algorithm to formalize the query is as shown in Figure 6.

```
Input:
Q= q1, ..., qk a query sequence
D-Ancestor B+ Tree, index of (symbol, prefix) pairs
S-Ancestor B+ Tree, index of (n, size) labels
DocId B+ Tree, mapping between the n values in node labels and document IDs.
Output: all occurrences of Q in the XML data
Search (<0,size>,1);  /*<0,size> is the label of the root node of the suffix tree */

Function Search (<n,size>, i)
If i<=|Q| then
        T <- retrieve, from the D-Ancestor B+ Tree, the S-Ancestor B+ Tree that represents qi;
        N<- retrieve from T, the S-Ancestor B+ Tree, all nodes with range inside (n,n+size];
        For each node c in N do /* assume c is labeled <n',size'> */
                Search (<n', size'>, i+1);
        End
Else
        Perform a range query [n, n+size] on the DocId B+ tree to output all document IDs in that
range;
End
```

Figure 6: Algorithm of Search for Query Processing

2.2 Others

APEX [14] is an adaptive path index for XML data. It utilizes the data mining algorithm to summarize frequently-used paths in the query workload. It doesn't keep all paths from the root and guarantees to maintain all paths of length two so that any label path query can be evaluated by join operations.

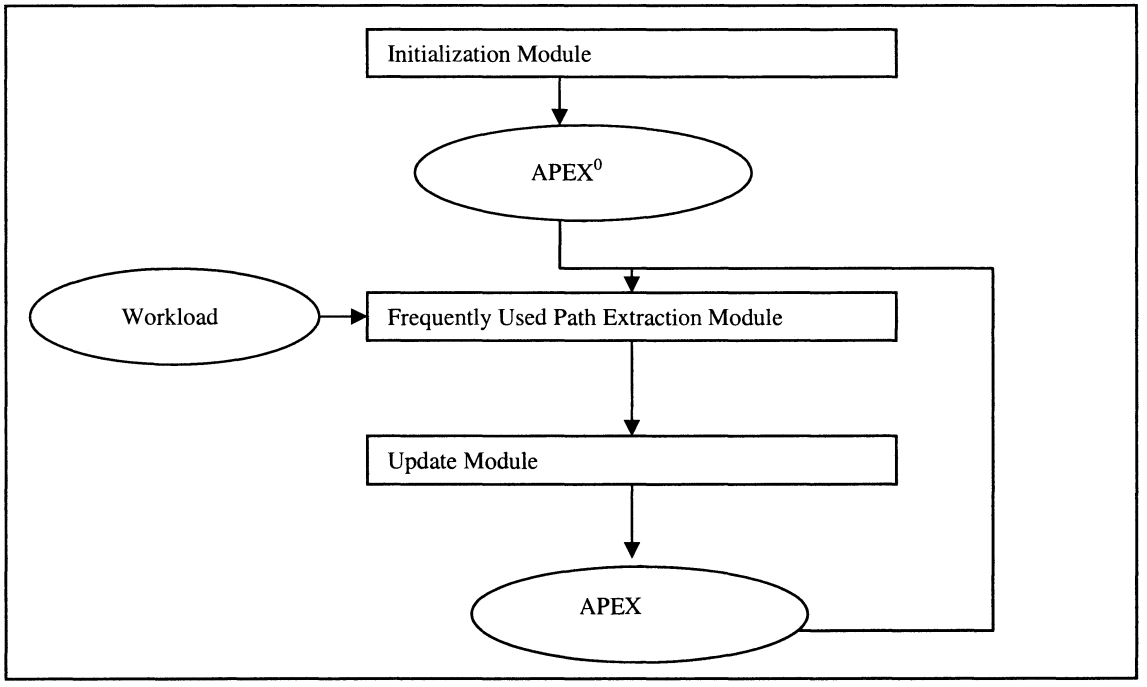

Figure 7: Core Architecture of Kweelt

The initialization module is invoked without query workloads only when APEX is built first. This module generates $APEX^0$ that is the simplest form of APEX and is used as a seed to build a more sophisticated APEX. When the workload is collected, the frequently used paths are extracted using data mining algorithm and the current APEX can be updated dynamically into a more detailed version of APEX. APEX is very efficient for queries with a simple path expression that is a sequence of labels starting from the root of the XML data. It also can be incrementally updated in order to minimize the overhead of construction if there is a change for query workload.

In our project, instead of generating frequently used paths as done in APEX using data mining algorithm, we can easily determine the frequently used searches. Since it is a bibliography document, most end users search for author or title. That is one of motivations that we decide to build separate B+ tree for author and title. Therefore if a dataset is changed, we don't need to scan the entire the data set to update frequently used paths as in APEX. We can check the nature and structure of this dataset to decide the frequently used searches and build corresponding B+ tree for that.

DataGuides [15] is a system to enable query formulation and optimization in semistructured databases. A strong DataGuide can also serve as a path index to solve the problem of creating and maintaining a path index without a fixed schema. However it restricted to raw paths and does not support complex path expression with combination of regular expression queries [15]. As DataGuides, our project doesn't support all the XML queries. However it supports some information retrieval search that is very critical for bibliography documents and provides very good performance for frequently used simple queries.

3. Designs and Implementation

Bibliography dataset is a large set of small documents. XML-Based bibliography database contains a large set of small XML documents that share the same schema. The database we used for benchmark is DBLP. DBLP provides the bibliographic information on major computer science journals and proceedings. It is one of most widely-used bibliography databases.

Our project is to build index and support the efficient information retrieval search and query from the bibliographic database. In order to achieve this objective, we implement three B+ trees: DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree.

All the data are stored as key/value pair in these three B+ trees. The key is stored as an index record in internal page in lexicographic sorted order. The corresponding value is stored as a real record in leaf page in lexicographic sorted order. These three B+ trees share the similarities while they do have the difference in key structures. The major difference of these three B+ trees is shown in the different representation of internal pages.

DBLP B+ tree uses the target path sequence as a key and DocID as a value. So it supports the query with the format of target path sequence.

For DBLP Author B+ tree, it uses the last name plus the first character of first name of each author as a key and DocID as a value. So it supports the query searching for last name plus first character of first name of an author or last name of an author only.

For DBLP Title B+ tree, it uses a keyword from a set of keywords we generate for each title as a key and DocID as a value. So it supports the query searching for keyword from the title.

Our choice of above three indices is based on the way bibliography documents are accessed. When the user tries to access to a very large bibliography database, he may enter the full path to locate the document or only partial path. DBLP B+ tree can achieve all these goals. However bibliography is a list of documents related to a subject. Most users only are intended to search for specific author or some title keyword. Based on this fact, we developed DBLP Author B+ tree and DBLP Title B+ tree from DBLP B+ tree. DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree not only provide the database storage but also support the information retrieval search.

We use Java for all the implementations. First we parse each XML document into sequence of strings. Based on different record structure, we can generate the target path sequences to construct DBLP B+ tree, or record the author last name with first character of first name to construct DBLP Author B+ tree or generate the keyword for each title to construct DBLP Title B+ tree. Then we parse the query to search for desired results from one of these three B+ trees.

3.1 Parsing and Transformation

The first step is use SAX to parse the XML document into the string of sequences. We use SAX for parsing since it is faster and use less memory than other parsing methods. This feature is very important since we handle the large dataset.

3.1.1 String Sequence of XML Data

DBLP is a list of writings related to the field of computer science. Figure 8 shows a sample document from DBLP. It contains 4 different elements: author, title, year and school.

```
<mastersthesis>
   <author>Kurt P.Brown</author>
   <title>PRPL: A Database Workload Specification Language, v1.3.</title>
   <year>1992</year>
   <school>Univ. of Wisconsin-Madison</school>
</masterthesis>
```

Figure 8: Sample Document from DBLP

Consider the sample document in Figure 8. We can present it using tree structure that is displayed in Figure 9. The depth for this XML document tree is 3. When the tree structure is generated for this document, each text value in this XML document becomes a leaf. There are 4 text values. Therefore the tree has 4 paths leading to corresponding leaf.

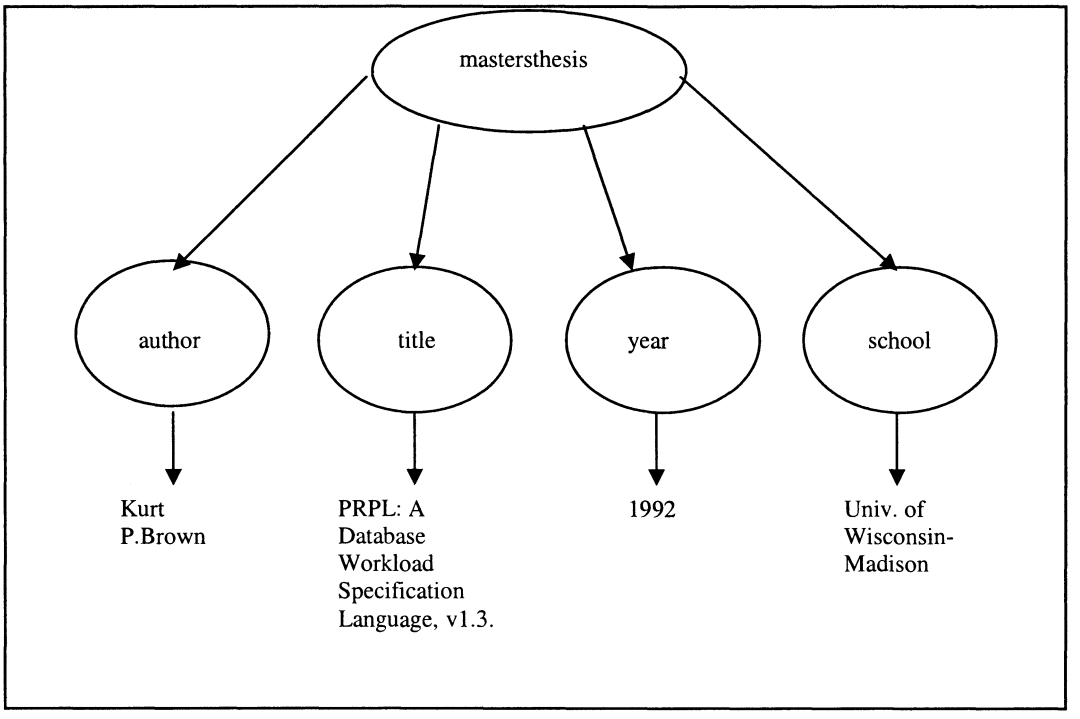

Figure 9: Tree structure of XML document

Using SAX we parse this XML document into sequence of strings that maintains its structure and data. Consider the XML document given in Figure 8. We can do preorder tree traversal of this XML document. Then it generates the sequence of strings that is shown in Figure 10.

masterthesis
author Kurt P. Brown
title PRPL: A Database Workload Specification Language,v1.3.
year 1992
school Univ.of Wisconsin-Madison.

Figure 10: Sequence of Strings Generated from Preorder Tree Traversal of Sample XML document

For DBLP B+ tree, we reconstruct sequence of strings into target path sequence to record all the ancestors for a context node.

3.1.2 Target Path Sequence

Definition 1: Target Path Sequence

A target path sequence, reconstructed from preorder tree traversal of an XML document, is a list of nodes, the context node with its ancestors. If the current target is a tag name, then the target path sequence is current tag name plus all its ancestors' tag name. Or if the current target is text value, then the target path sequence is this text value plus all its ancestors.

In Figure 11, it shows how we construct the target path sequence for each tag and text value in the XML document given in Figure 8. We use ^ to indicate the separator between two different nodes.

mastersthesis
author^mastersthesis
Kurt P. Brown^author^mastersthesis
title^mastersthesis
PRPL: A Database Workload Specification Language, v1.3.^title^mastersthesis
1992^year^mastersthesis
Univ. of Wisconsin-Madison^school^mastersthesis

Figure 11: Target Path Sequences

We also made assumption that all the queries have been encoded into the format of target path sequence.

3.2 Representation of Internal Page and Leaf Page

DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree have the similar structures of internal page and leaf page. Figure 12.1 and 12.2 shows the graphic representation of internal page and the representation of leaf page.

Each internal page contains many index records each of which is a key and pointer to an index record.

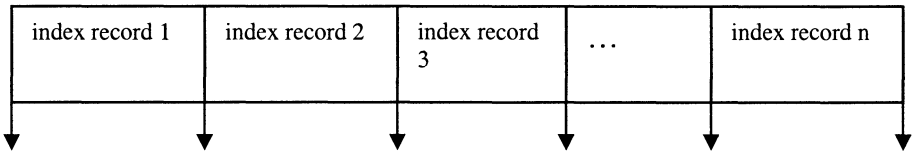

Figure 12.1: Representation of an Internal Page for DBLP B+ Tree

As shown in Figure 12.1 this sample internal page contains n keys with n+1 pointers. Each leaf page contains many data records each of which is a data item corresponding to a key in internal page.

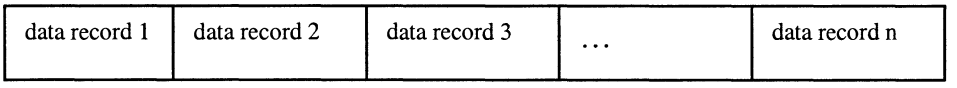

Figure 12.2: Representation of a Leaf Page for DBLP B+ Tree

For DBLP B+ tree, the key is the entire path up to current target, i.e. current target label plus all the ancestors' tags before it in preorder tree traversal order. Figure 13 shows the representation of an index record for DBLP B+ tree.

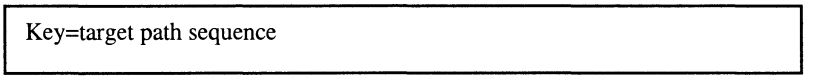

Figure 13: Representation of an Index Record for DBLP B+ Tree

For DBLP Author B+ tree, the key is last name plus the first character of first name that are generated from DBLP author tag. Figure 14 shows the representation of an index record for DBLP Author B+ tree.

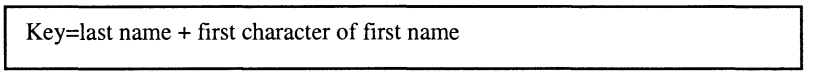

Figure 14: Representation of an Index Record for DBLP Author B+ Tree

For DBLP Title B+ tree, the key is each of keywords that are generated from DBLP title tag. The algorithm to generate the keyword from the title is very simple. We choose a set of strings which contains all the unimportant words defined by users. We insert all the unimportant words into junk set. If there is a match between a word in the title and a word in junk set, we discard this word. Otherwise we create an entry in DBLP Title B+ tree for this word. There is no limitation for the way to generate the keywords. It can be done using any other keyword generation tools. Figure 15 shows the representation of an index record for DBLP Title B+ tree.

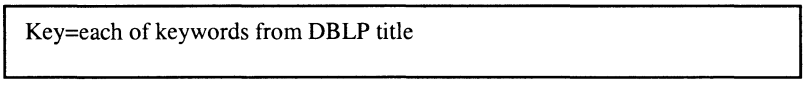

Figure 15: Representation of an Index Record for DBLP Title B+ Tree

From Figure 13, Figure 14 and Figure 15, we can found the key for each of these three B+ tree is different. The value for these three trees is same, which is DocID. DocID is a unique number assigned to each document in DBLP. For each key, we store the corresponding value that is DocID in a real data in the leaf page. Figure 16 shows the representation of a real record for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree.

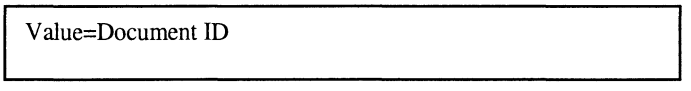

Figure 16: Representation of a Real Record for DBLP B+ Tree

3.3 Storing Multiple DocIDs for the Duplicate Keys

In the implementation of all our three B+ trees, we support duplicate key. Because a lot of XML documents share the same tag name. That means one single path can appear in many documents. One of the features of Berkeley DB is to support the creation of multiple data items for a single key item. By default, multiple data items are not permitted so the store operation will overwrite the previous data item with this new coming data item for this key. In order to solve this problem, we can configure the Berkeley DB system to support the duplication of keys. After we configure the Berkeley DB in this way, only one copy of the key will be stored for each set of multiple data items.

Since the height of B+ tree is the single most important measure of its performance of the tree, a B+ tree with smaller height is desirable. The greater the fanout is, the smaller the height of the tree is. Under such a configuration of Berkeley DB system, the space is saved and more keys can be stored in internal page of B+ tree that means the fanout of B+ tree becomes greater. So this makes the implementation of all our three B+ trees desirable which will eventually improve the performance.

3.4 2-Tier and 1-Tier B+ Tree

The 2-tier B+ tree represents a more traditional approach where there are two levels of B+ trees. The first level of B+ tree records all the keys. The second level of B+ tree is a set of B+ trees each of which records the list of corresponding values for each key. Both levels are in B+ tree structure. DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree we build are 2-tier B+ trees.

For testing purpose, we also build 1-tier B+ tree for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree. The key in 1-tier B+ tree is appended with DocID that makes each key unique. Therefore for each key there is only one value corresponds to it. There is no need for the second level organization.

When we refer to DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree, we mean 2-tier B+ trees we build unless we mention it is 1-tier B+ tree explicitly. Since the comparison of 2-tier and 1-tier is not our main objective, we describe it in appendix part about its key/value structure, and performance. The performance of 1-Tier B+ tree is worse than 2-Tier B+ tree. The partial reason we thought might be compression of keys. Since Berkeley DB compression algorithm provides a good compression for 2-Tier B+ trees while it doesn't support the compression for 1-Tier B+ tree, it caused more overheads for 1-Tier B+ tree which might potentially affect the query performance of 1-Tier B+ tree.

4. Benchmarks

4.1 Test Environment

We use PIII computers with 256MB memory, 30GB hard disk and XP operating system to conduct all the experiments. The details about the environment displayed in table 1:

Processor	Intel(R) Pentium(R) III Mobile CPU 1000MHz
Memory	256MB
Operating System	Microsoft Windows XP Professional
Operating System Version	5.1.2600
Hard Disk	30GB

Table 1: Environment for the experiments

4.2 Assumptions

a) Bibliography document is fixed.

The bibliography document we use for experiments is DBLP [17]. DBLP stands for Digital Bibliography Library Project. DBLP provides the bibliographic information on major computer science journals and proceedings. Each record of DBLP corresponds to a publication that contains the information about author, title, publisher and etc. Each record of DBLP can be one of articles, inproceedings, proceedings, books, incollections, phd theses, master theses or URLs. All these information can be found in DTD file associated with DBLP.xml file.

DBLP is widely used to benchmark XML indexing methods. The statistics for the version we downloaded is shown in Table 2.

Number of Records	487,226
Number of Elements	4,967,198
Number of Attributes	1,051,414
Size of Data	205,838KB

Table 2: Statistics for DBLP Dataset

b) Mapping between DocID and PageID exists.

DocID is a unique number for each document. We assume there is a mapping between DocID and PageID. Therefore DocID is actually a pointer pointing to the page that the document resides on.

4.3 Benchmark Experiments

Figure 17 shows the index size for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree.

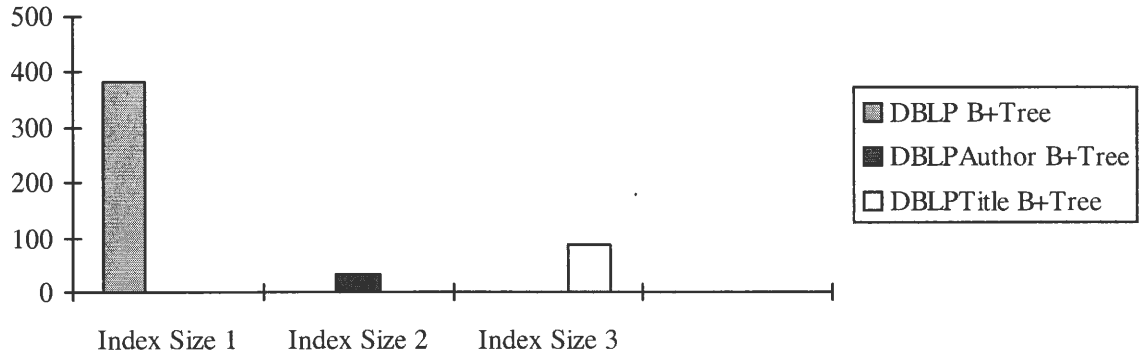

Figure 17: B+ Tree Index Size (MB)

Figure 18 shows the average index construction times for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree under the same environment with 256KB cache size.

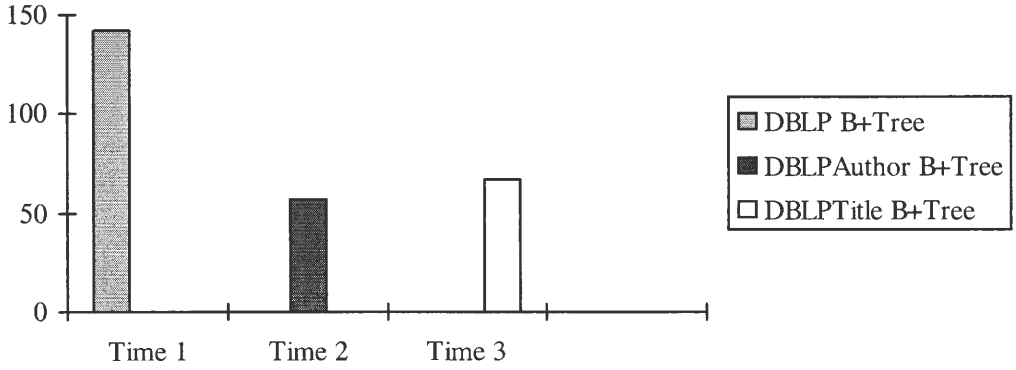

Figure 18: B+ Tree Average Construction Time (Minute)

Table 3 shows the statistics for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree.

	DBLP B+ Tree	DBLP Author B+ Tree	DBLP Title B+ Tree
Number of Entries	8,607,881	1,093,097	3,024,414
Number of Unique Keys	1,714,249	211,898	238,204
Size of B+ Tree	384.316MB	33.936MB	89.328MB

Table 3: Statistics for DBLP B+ Tree, DBLP Author B+ Tree and DBLP Title B+ Tree

Table 4 shows the statistics about the page and cache for DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree.

	DBLP B+ Tree	DBLP Author B+ Tree	DBLP Title B+ Tree
Size of Page	4KB	4KB	4KB
Number of Internal Page	1731	119	1629
Number of Leaf Page	61,471	7,180	7,343
Level of B+ Tree	4	3	3
Size of Cache	256KB	256KB	256KB

Table 4: Query Statistics for DBLP B+ Tree, DBLP Author B+ Tree and DBLP Title B+ Tree

Based on the statistics shown in Table 4, we give the result of sample queries in Tables 5, Table 6 and Table 7.

We use 3 different types of query sets to test the performance of DBLP B+ tree. The first type is path expression stating the exact path for the target without omitting any middle path. The second type is path expression with * symbol and * symbol selects all the children of the context node. The third type is path expression with // symbol and // symbol selects all the descendants of the context node.

The sample queries are as follows:
Sample Q1 from first type query set: /article/author
Sample Q2 from second type query set: */year/1999
Sample Q3 from third type query set: //year/1994

Table 5 shows the query performance of three different types of sample queries for DBLP B+ tree.

Query	Time
"/article/author"	9423ms
"*/year/1999"	5288ms
"//year/1994"	3385ms

Table 5: Sample Query Time for DBLP B+ Tree

There are two different types of queries for DBLP Author type. It can search for the last name only or search for last name plus first character of first name for DBLP Author B+ tree. The sample queries are as follows:
Sample Q1 searching for last name: Zhang J
Sample Q2 searching for last name plus first character of first name: Brown

Table 6 shows the query performance for the sample queries.

Query	Time
"Zhang J"	50 ms
"Brown"	140ms

Table 6: Sample Query Time for DBLP Author B+ Tree

There is only one type of query for DBLP Title B+ tree. It supports the search for the keyword from the title. The sample query is as follows:

Sample query searching for the keyword from the title: database

Table 7 shows the query performance of single keyword retrieval for DBLP Title B+ tree.

Query	Time
"Database"	906ms

Table 7: Sample Query Time for DBLP Title B+ Tree

The default cache size for test is 256KB. In order to show the experiments benefit from the increased cache size. We also run several tests with increased cache size for each B+ tree. We choose to use three different cache sizes: 256KB, 4MB and 8MB. We construct 2 charts for each B+ tree to indicate the difference of construction time and query time. It helps in displaying the positive difference that increased cache size makes. As shown from each chart, the overall trend is that when cache size increase, both the construction of B+ trees and query becomes more efficient. The performance is associated with the size of data set. If the size is very large, the increased cache size can improve the performance quite a bit. Otherwise the improvement is not quite visible.

For DBLP B+ tree, the result based on three cache sizes is shown in Table 8.1, Table 8.2 and Table 8.3.

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"/article/author"	256KB	8490258ms	9423ms
"/article/author"	4MB	8018169ms	9293ms
"/article/author"	8MB	7909453ms	9203ms

Table 8.1: Result for DBLP B+ Tree Sample Query 1

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"*/year/1999"	256KB	8490258ms	5288ms
"*/year/1999"	4MB	8018169ms	5098ms
"*/year/1999"	8MB	7909453ms	5017ms

Table 8.2: Result for DBLP B+ Tree Sample Query 2

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"//year/1994"	256KB	8490258ms	3385ms
"//year/1994"	4MB	8018169ms	3295ms
"//year/1994"	8MB	7909453ms	3134ms

Table 8.3: Result for DBLP B+ Tree Sample Query 3

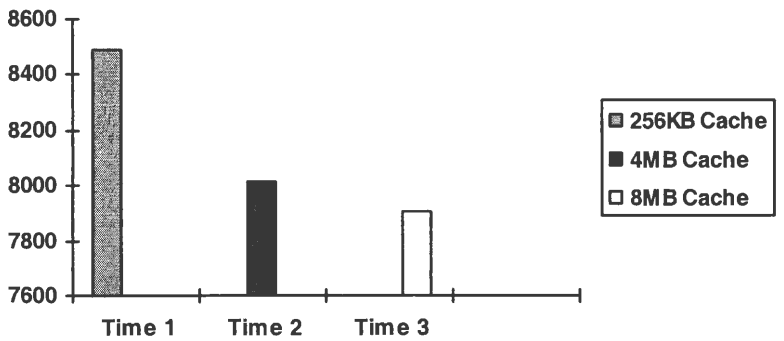

Figure 19: Result for DBLP B+ Tree Construction Time (Seconds)

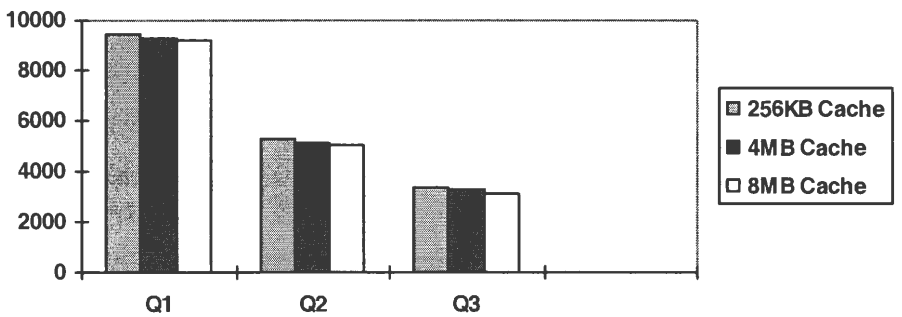

Figure 20: Result for DBLP B+ Tree Three Sample Queries (MS)

For DBLP Author B+ tree, the result based on three cache sizes is shown in Table 9.1 and Table 9.2.

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"Zhang J"	256KB	3407730ms	50ms
"Zhang J"	4MB	3387571ms	40ms
"Zhang J"	8MB	3348595ms	38ms

Table 9.1: Result for DBLP Author B+ Tree Sample Query 1

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"Brown"	256KB	3407730ms	140ms
"Brown"	4MB	3387571ms	132ms
"Brown"	8MB	3348595ms	130ms

Table 9.2: Result for DBLP Author B+ Tree Sample Query 2

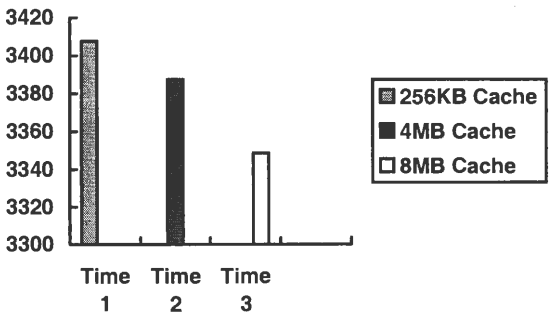

Figure 21: Result for DBLP B+ Tree Construction Time (Seconds)

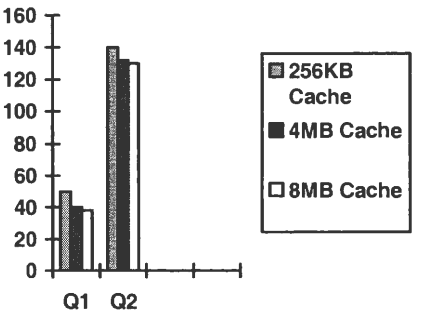

Figure 22: Result for DBLP B+ Tree Construction Time (MS)

For DBLP Title B+ tree, the result based on three cache sizes is shown in Table 10.

Sample Query	Cache Size	B+ Tree Construction Time	Query Time
"Database"	256KB	4001965ms	906ms
"Database"	4MB	3836696ms	872ms
"Database"	8MB	3768810ms	851ms

Table 10: Result for DBLP Title B+ Tree Sample Query

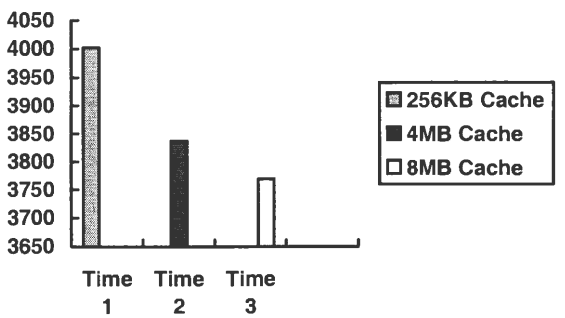

Figure 23: Result for DBLP Title B+ Tree Construction Time (Seconds)

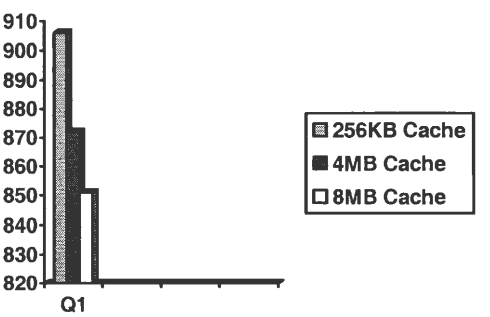

Figure 24: Result for DBLP Title B+ Tree Query (MS)

5. Discussions and Future Work

We have build three B+ trees and plan to investigate the possibility of integration with other query tool or other storage architecture. We will continue to exploit the performance of this indexing method for other bibliography XML-based datasets.

5.1 The Storage Issue

In this manuscript we present three types of B+ trees. For each B+ tree, we use DocID to indicate the corresponding document and assume the document can be retrieved based on this DocID. Actually DocID should be a pointer to the position where the corresponding document is on the disk in order to achieve this purpose. This requires we need further work on storage issue. We propose to create mapping between the DocID and the position of corresponding document on the disk and record the information in a table like structure. Then with this DocID on hand, we can find the corresponding position based on this mapping and retrieve the document efficiently.

5.2 Integration with Kweelt

For DBLP B+ tree, we assume the query is given as the encoded target path expression. We did not implement a query parser. Thus, we don't parse queries and optimize it. In the future we can allow DBLP B+ tree to support more queries. We also can use B+ trees we construct and combine it with Kweelt to support X-Query. Kweelt offers multiple XML back-ends. The query evaluator does not impost any specific storage for XML but relies on a set of interfaces (Node and NodeList) implemented by a NodeFactory. Kweelt comes with a couple of build-in node factories (DOM, DOM+, SAX, Wizdom), but the user can easily provide their own node factory [18]. Figure 25 shows the architecture of Kweelt [18].

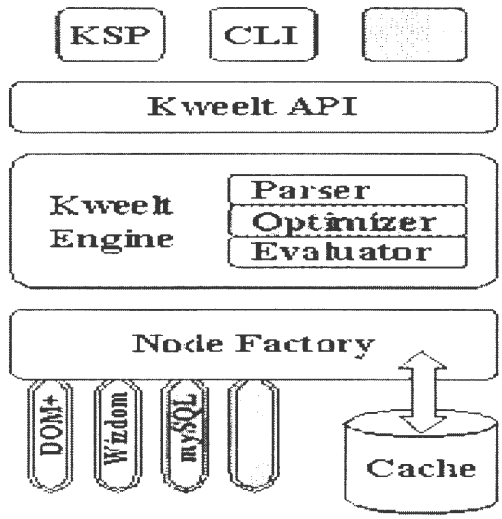

Figure 25: Core Architecture of Kweelt

We can see the node management is delegated to the node factory. Nodes can be created by a specific implementation and Kweelt doesn't need to know about it. This flexibility provides us the freedom to build our own storage and still fully use Kweelt based on our own storage.

5.3 Branch Queries

DBLP B+ tree is not efficient in dealing with queries that contains branches. It needs to disassemble a query into several multiple sub queries and join the results of sub queries into the final output. The join operation is expensive and affects the query performance. In order to achieve the better performance the join operations should be avoid as much as possible.

However, for DBLP, each document is small. Its tree structure has the depth at most 6 and most depth is only 3. Therefore if the query containing branches is given we still can query the XML documents based on the deepest sub query and return all the documents containing this pattern without worrying about the other sub queries. In this way instead of getting subtree of XML documents we can get all the entire documents. But the performance is still good. Since the depth of tree structure of each document is at most 6 and most of them are only 3. The queries containing branches are very rare and the performance is not affected much.

6. Conclusion

Instead of working on all general purpose XML data, we focus on the query and indexing of large xml-based bibliography database - DBLP.

We have parsed the XML data into sequence of strings and constructed DBLP B+ tree, DBLP Author B+ tree and DBLP Title B+ tree for large DBLP bibliography xml-based databases. These are 2-tier B+ trees. DBLP B+ tree supports the query in the format of target path expression. DBLP Author B+ tree supports the query to search for last name only or last name with first character of first name. DBLP Title B+ tree allows the user to search for a single keyword from the title. All these functions provide the user with a good opportunity to search and retrieve the information from a large xml-based bibliography database very efficiently. We have conducted many experiments to test the performance of construction time for each B+ tree and to observe the query efficiency. The benchmark is given for each B+ tree construction and several sample queries.

We also implemented 1-tier DBLP B+ tree, 1-tier DBLP Author B+ tree and 1-tier DBLP Title B+ tree. We did comparison between 1-tier B+ tree and 2-tier B+ tree in appendix. We conclude that given a large set of XML data, 2-tier B+ tree has achieved the better performance than 1-tier B+ tree since 2-tier B+ tree has more efficient compression and therefore less database size.

7. References

1. ISO 8879: ISO (International Organization for Standardization). ISO 8879: 1986(E). Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML). First edition – 1986-10-15. [Geneva]: International Organization for Standardization, 1986.
2. http://www.w3.org/XML/: Extensible Markup Language (XML). Accessed on January 16, 2004.
3. M. Daconta, A. Saganich. XML development with Java 2. Sams Publishing 2000.
4. http://www.w3.org/TR/2004/REC-xml-20040204/: Extensible Markup Language (XML) 1.0 (Third Edition). Accessed on February 20, 2004.
5. http://www.w3.org/TR/xpath: XML Path Language (XPath) Version 1.0. Accessed on January 22, 2004.
6. http://www.w3.org/TR/2002/WD-xquery-20020816/: XQuery 1.0: An XML Query Language. Accessed on February 6, 2004.
7. H. Katz. XQuery from the experts: a guide to the W3C XML query language. Addison-Wesley 2004.
8. R. Ramakrishnan, J. Gehrke. Database management systems third edition. Mc Graw Hill 2003.
9. P. O'Neil, E. O'Neil. Database principles, programming and performance second edition. Morgan Kaufmann Publisher.
10. S. Gadia. An elemental approach to databases in slides. Department of Computer Science, Iowa State University.
11. Sleepycat™ Software, Inc. Berkeley DB. New Riders Publishing 2001.
12. W.Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th VLDB Conference*, pages 212-223, Montreal, 1980.
13. H. Wang, S. Park, W. Fan, P. Yu. VIST: a dynamic index method for querying XML data by tree structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 110-121, June 2003.
14. C. Chung, J. Min, K. Shim. APEX: an adaptive path index for XML data. In *ACM SIGMOD*, June 2002.
15. R. Goldman, J. Widom. DataGuides: enable query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436-445, August 1997.
16. T. Milo, D. Suciu. Index structures for path expression. In *Proceedings of 7th International Conference on Database Theory*, pages 277-295, January 1999.
17. http://dblp.uni-trier.de/xml/: DBLP XML Dataset Download. Accessed on March 12, 2004.
18. http://kweelt.sourceforge.net/: Kweelt: Querying XML in the New Millennium. Accessed on February 18, 2004.

8. Appendix: Comparison between 2-Tier and 1-Tier B+ Tree

DBLP B+ tree, DBLP Author B+ tree and DBLP Author B+ tree we construct are all 2-tier B+ tree. We also build corresponding 1-tier B+ tree. 1-tier B+ tree still maintain the same internal page and leaf page structure. However, in 1-tier B+ tree, for each index record inside the internal page, we append DocID to the original key in the 2-tier B+ tree, which makes each key unique and the tree is 1-tier.

Table A1 shows the structure of key for DBLP B+ tree.

Key=target path sequence||DocID

Table A1: Representation of an Index Record for DBLP B+ tree

Table A2 shows the structure of key for DBLP Author B+ tree.

Key=last name + first character of first name||DocID

Table A2: Representation of an Index Record for DBLP Author B+ tree

Table A3 shows the structure of key for DBLP Title B+ tree.

Key=each of keywords from DBLP title

Table A3: Representation of an Index Record for DBLP Title B+ tree

Figure A1 shows the comparison between the size of 2-tier B+ trees and the size of 1-tier B+ trees.

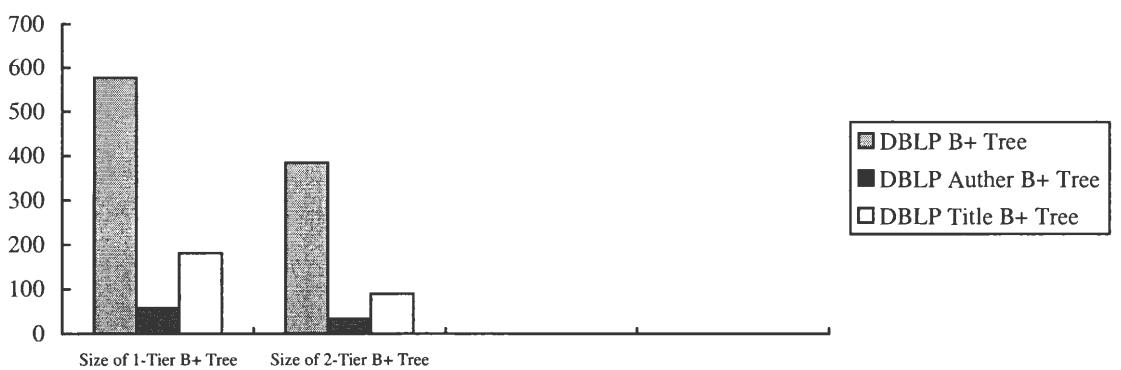

Figure A1: Comparison of Sizes of 2-Tier and 1-Tier B+ Trees (MB)

Figure A2 shows the comparisons between the construction time of 2-tier B+ trees and the construction time of 1-tier B+ trees.

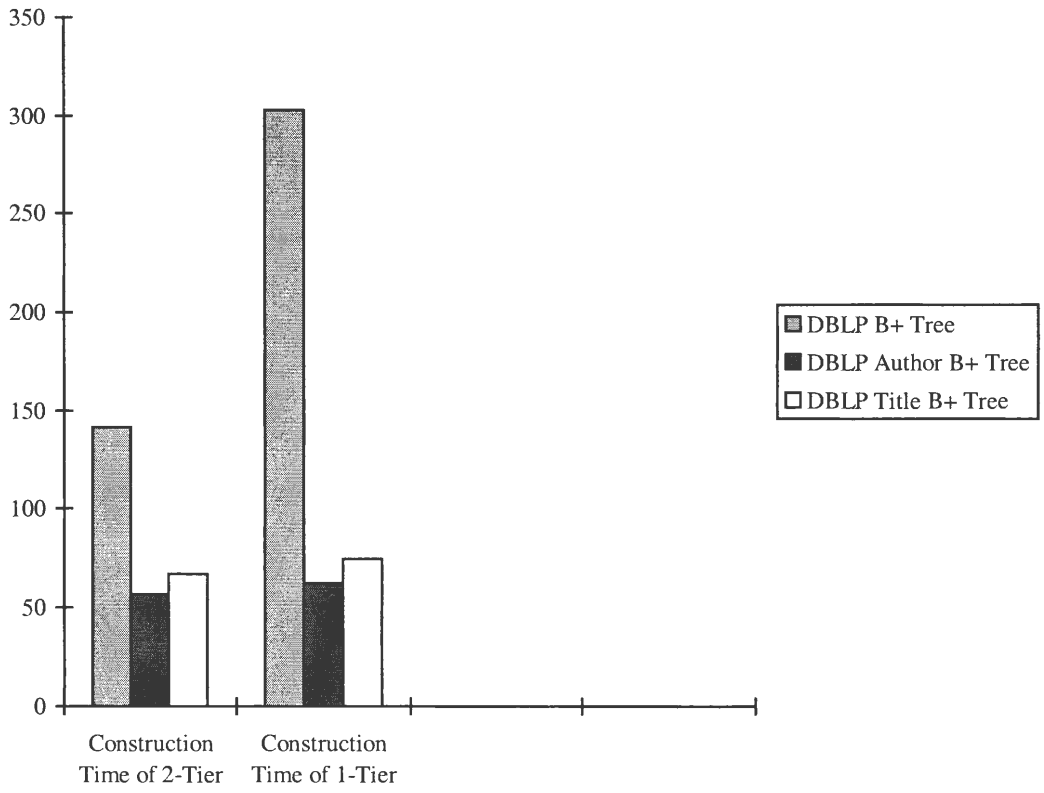

Figure A2: Comparison of Construction Times of 2-Tier and 1-Tier B+ Trees (Minute)

From Figure A1 and A2 we can find that 2-tier B+ tree achieve the better performance in terms of construction time than 1-tier B+ tree using the same testing environment with 256KB cache. The reason is that 2-tier B+ tree has very good compression and result in the smaller B+ tree size.

We use the same sample query set to compare 1-tier and 2-tier B+ trees under the same environment with increased cache size. From the following charts, it is very obvious that the 2-tier B+ tree can execute the queries much more efficient than 1-tier B+ trees under the same environment with the same cache size although the performance of 1-tier B+ tree is improved with increased cache size but still not as good as in 2-tier B+ tree under the same environment. For each sample query, the chart is built to show the large difference between 2-Tier and 1-Tier B+ tree. From the following charts, the

execution time of 1-Tier is almost double than the 2-Tier B+ tree. Therefore the performance of 2-Tier is much better under the same environment.

Table A4.1, A4.2 and A4.3 shows the result of sample query 1, sample query 2 and sample query 3 for 1-tier DBLP B+ tree with different cache sizes.

Sample Query	Cache Size	Query Time
"/article/author"	256KB	21661ms
"/article/author"	4MB	21270ms
"/article/author"	8MB	20479ms

Table A4.1: Query Time of 1-Tier DBLP B+ Tree with Different Cache Sizes

Sample Query	Cache Size	Query Time
"*/year/1999"	256KB	11847ms
"*/year/1999"	4MB	11617ms
"*/year/1999"	8MB	11476ms

Table A4.2: Query Time of 1-Tier DBLP B+ Tree with Different Cache Sizes

Sample Query	Cache Size	Query Time
"//year/1994"	256KB	7661ms
"//year/1994"	4MB	7591ms
"//year/1994"	8MB	7241ms

Table A4.3: Query Time of 1-Tier DBLP B+ Tree with Different Cache Sizes

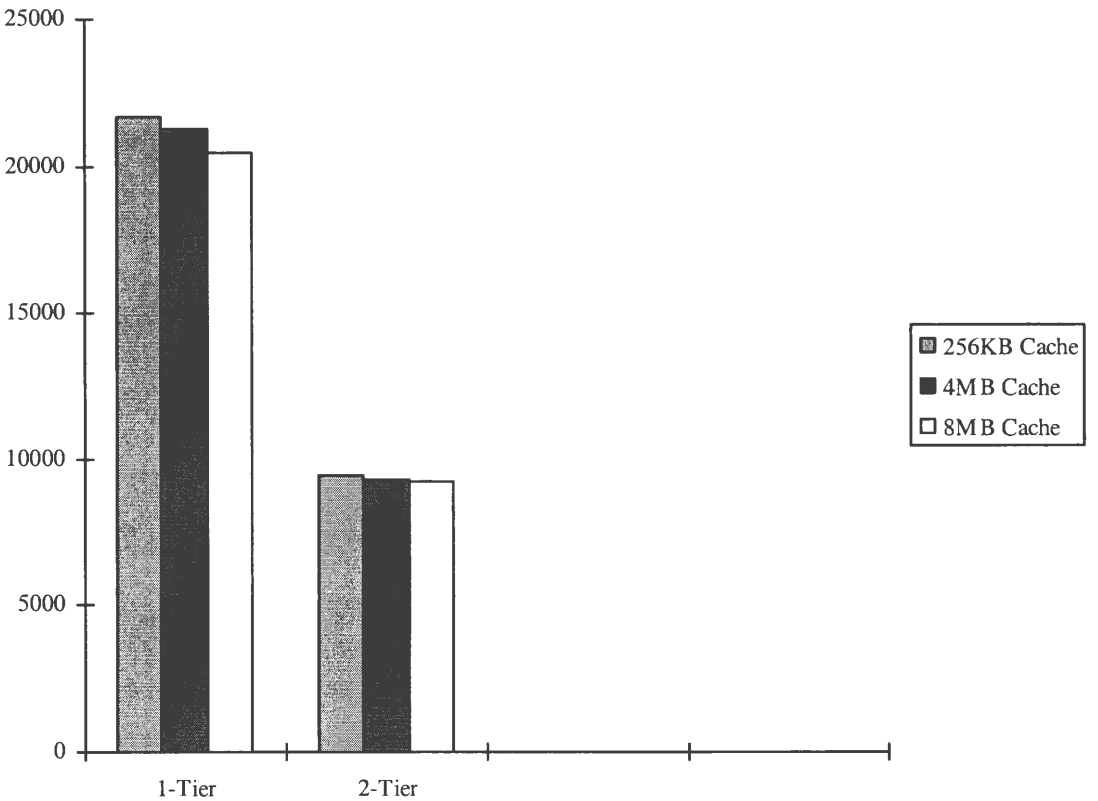

Figure A3.1: Comparison of Query Times of 2-Tier and 1-Tier DBLP B+ Trees for Sample Query 1 (MS)

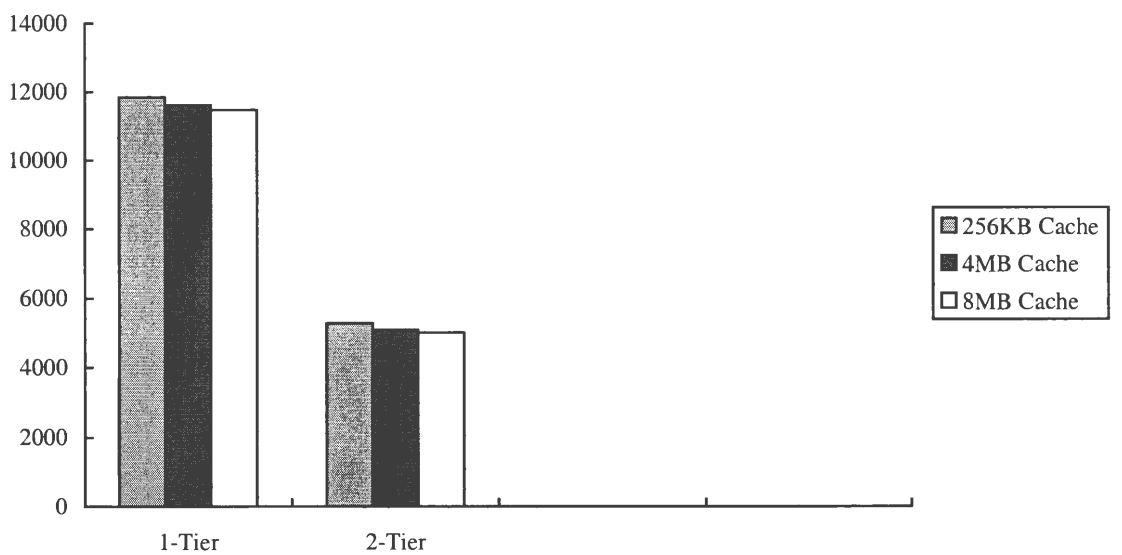

Figure A3.2: Comparison of Query Times of 2-Tier and 1-Tier DBLP B+ Trees for Sample Query 2 (MS)

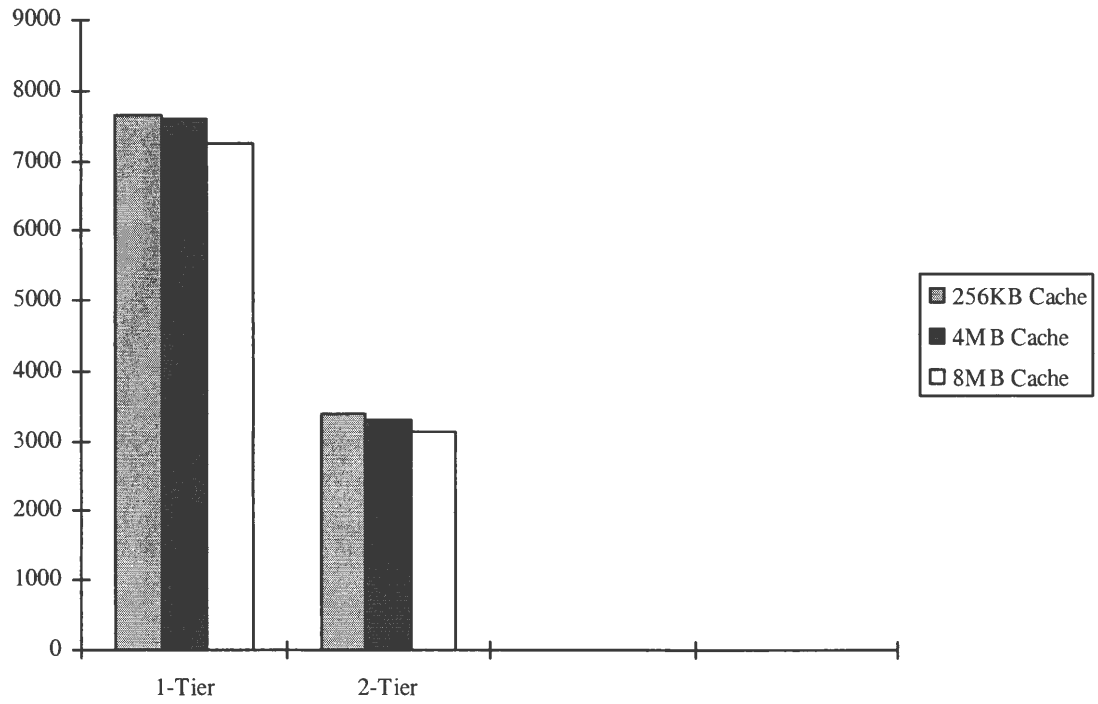

Figure A3.3: Comparison of Query Times of 2-Tier and 1-Tier DBLP B+ Trees for Sample Query 3 (MS)

Table A5.1 and A5.2 show the results of sample queries for 1-tier DBLP Author B+ tree with different cache sizes.

Sample Query	Query Time	Cache Size
"Zhang J"	812ms	256KB
"Zhang J"	621ms	4MB
"Zhang J"	581ms	8MB

Table A5.1: Query Time of 1-Tier DBLP Author B+ Tree with Different Cache Sizes

Sample Query	Query Time	Cache Size
"Brown"	2616ms	256KB
"Brown"	2501ms	4MB
"Brown"	2415ms	8MB

Table A5.2: Query Time of 1-Tier DBLP Author B+ Tree with Different Cache Sizes

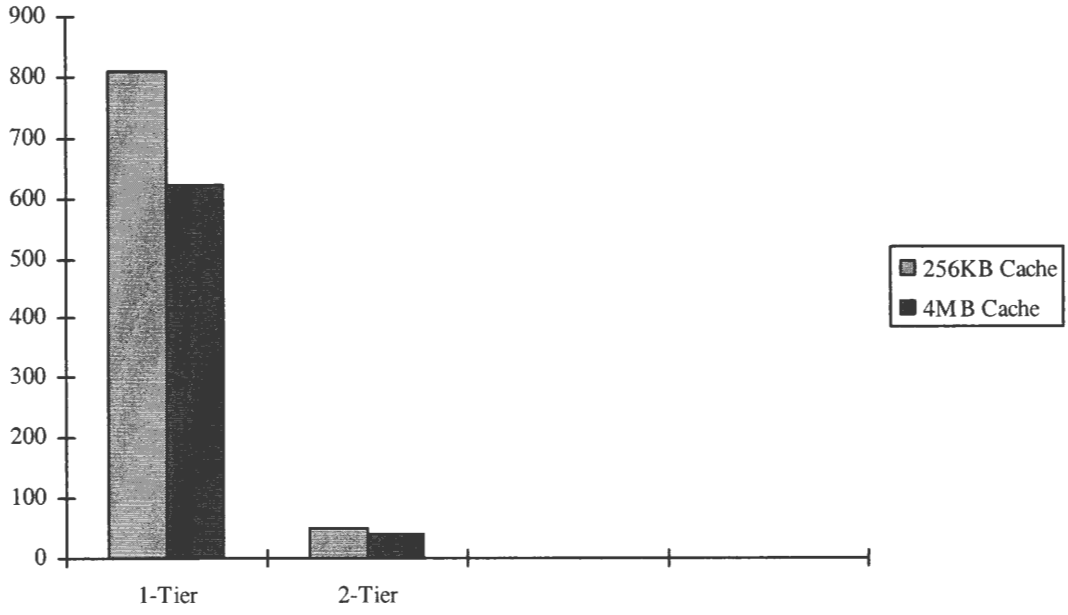

Figure A4.1: Comparison of Query Times of 2-Tier and 1-Tier DBLP Author B+ Trees for Sample Query 1 (MS)

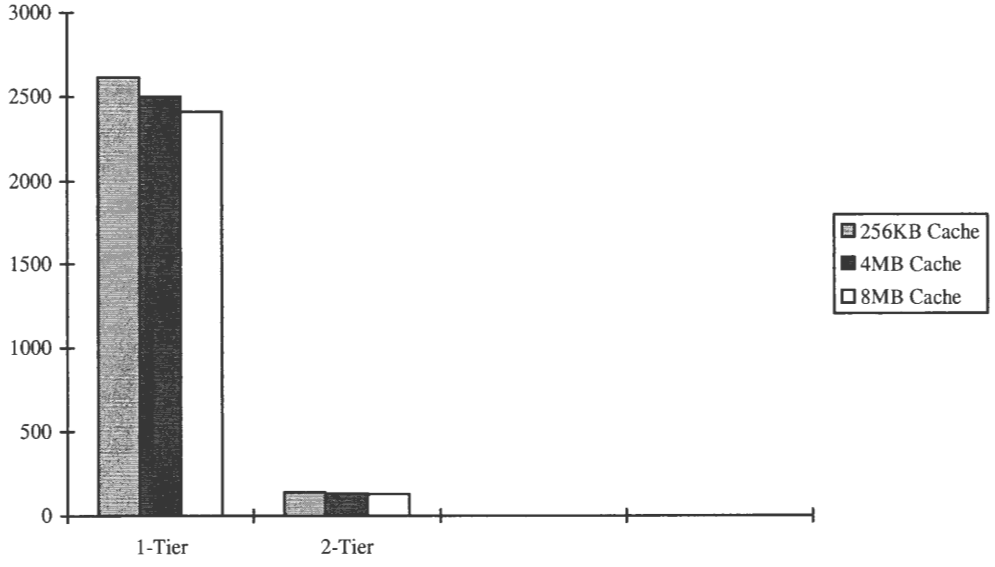

Figure A4.2: Comparison of Query Times of 2-Tier and 1-Tier DBLP Author B+ Trees for Sample Query 2 (MS)

Table A6 shows the result of sample queries for 1-tier DBLP Title B+ tree with different cache sizes.

Sample Query	Query Time	Cache Size
"Database"	1522ms	256KB
"Database"	1302ms	4MB
"Database"	1032ms	8MB

Table A6: Query Time of 1-Tier DBLP Title B+ Tree with Different Cache Sizes

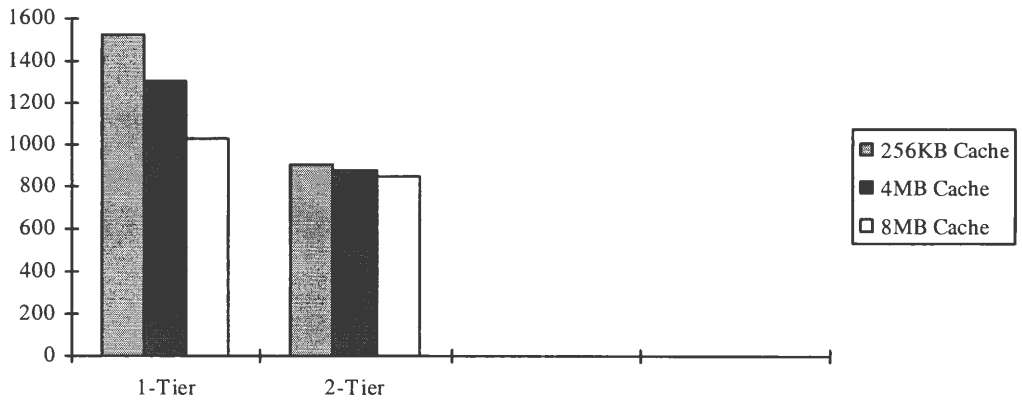

Figure A5: Comparison of Query Times of 2-Tier and 1-Tier DBLP Title B+ Trees for Sample Query (MS)